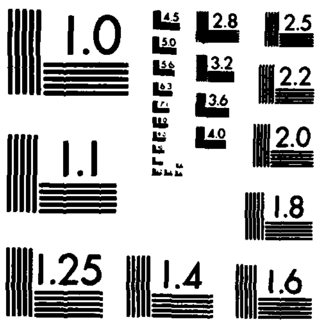
AD-A146 887 PROGRAM VISUALIZATION: GRAPHICS SUPPORT FOR SOFTWARE DEVELOPMENT(U) COMPUTER CORP OF AMERICA CAMBRIDGE MA C F HEROT ET AL. 28 SEP 84 N00014-81-C-0456 1/1

UNCLASSIFIED F/G 9/2 NL

END


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

12

AD-A146 887

Contract N00014-81-0456; NR 049-495

PROGRAM VISUALIZATION: GRAPHICS SUPPORT FOR SOFTWARE DEVELOPMENT

Christopher F. Herot
Gretchen P. Brown
Richard T. Carling
David A. Kramlich

Computer Corporation of America
4 Cambridge Center
Cambridge, Massachusetts 02142

28 September 1984

Final Report

Prepared for

Defense Advanced Research Projects Agency
Defense Sciences Office
Systems Sciences Division

OFFICE OF NAVAL RESEARCH
800 N. Quincy Street
Arlington VA 22217

84 10 18 014

Contract N00014-81-0456; NR 049-495

PROGRAM VISUALIZATION: GRAPHICS SUPPORT FOR SOFTWARE DEVELOPMENT

Christopher F. Herot
Gretchen P. Brown
Richard T. Carling
David A. Kramlich

Computer Corporation of America
4 Cambridge Center
Cambridge, Massachusetts 02142

28 September 1984

Final Report

Prepared for

Defense Advanced Research Projects Agency
Defense Sciences Office
Systems Sciences Division

OFFICE OF NAVAL RESEARCH
800 N. Quincy Street
Arlington VA 22217

Unclassified
Security Classification

AD-A146 887

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*

Computer Corporation of America
Four Cambridge Center
Cambridge, MA 02142

2a. REPORT SECURITY CLASSIFICATION

Unclassified

2b. GROUP

-

3. REPORT TITLE

PROGRAM VISUALIZATION:

GRAPHICS SUPPORT FOR SOFTWARE DEVELOPMENT

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

Final Report

5. AUTHOR(S) *(First name, middle initial, last name)*

Christopher F. Herot, Gretchen P. Brown, Richard T. Carling, David A. Kramlich

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
28 September 1984	38	17

8a. CONTRACT OR GRANT NO.

N00014-81-0456

b. PROJECT NO.

NR049-495

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. DISTRIBUTION STATEMENT

Distribution of this document is unlimited

11. SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY

Defense Advanced Research Projects Agency
Defense Sciences Office
Systems Sciences Division

13. ABSTRACT

This document is the final report on the design and implementation of a program visualization (PV) environment, intended to offer the user an integrated graphics programming support system. The PV environment has capitalized on recent progress in the graphical representation of information, to provide designers and programmers with both static and dynamic (animated) views of systems. The PV research prototype supports programming in C, although large portions of the system are independent of the software development language.

p. 1

DD FORM 1 NOV 65 1473 REPLACES DD FORM 1473, 1 JAN 64, WHICH IS OBSOLETE FOR ARMY USE.

Unclassified
Security Classification

Unclassified
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
programming workstations, graphics, integrated environments, software engineering, computer animation, program instrumentation						

Unclassified
Security Classification

CONTENTS

1. INTRODUCTION

Some programs are so simple and so unimportant that they can be conceived, developed, used, and possibly thrown away in a single sitting at an interactive computer terminal. The art of conversational programming has been developed to facilitate such expression.

However the bulk of programs upon which society relies are complex. They have been developed and refined by many individuals working over many years. They may not now be understood by any single person -- they may never have been understood by any single person.

The goal of program visualization is to facilitate the understanding of programs by people. To visualize means ~~to~~ see or form a mental image of. Successful program visualizations aid programmers in the formation of clear and correct mental images of the structure and function of programs.

Program visualization can be useful in all of the stages of a program's lifecycle:

1. in listing the *requirements* that the program must satisfy;

2. in specifying the *design* of a software system to meet the requirements;

3. in carrying out the *coding* of the system following the plan of the design;

4. in *testing* and *debugging* the code, to guarantee that it conforms to the design and fulfills the requirements;

5. in the *maintenance* of the system, to keep it functioning despite changes in the requirements and the discovery of new bugs; and

6. in helping the end-user *use* the program by showing how it operates and how it arrived at the results it presents.

The challenge of a research program in program visualization (abbreviated here as "PV") is to encompass these separate phases of the software development process within a unified conceptual framework in a way which benefits rather than burdens the user at each stage in the program's lifecycle.

This document is the final report for ONR Contract Number N00014-81-C-0456, covering work from May 1981 through October 1983. In addition to the authors, contributers to the project included Paul Souza (WGBH TV), Rebecca Allen (New York Institute of Technology), Ronald M. Baecker (Human Computing Resources Corporation), and Aaron Marcus (Aaron Marcus and Associates).

The Program Visualization project included both research and system-building components. Section 2 enumerates different types of program visualizations, listing the visualization types supported in the PV system prototype. Section 3 gives an overview of the PV environment, identifying four classes of capabilities that are discussed in the four sections (4-7) that follow it. Section 8 describes one solution to an important technical problem in program visualization, the need to monitor large running programs in order to detect the updates to be displayed.

A copy of a videotape showing the PV research prototype in operation has been submitted as a supplement to this report.

2. CATEGORIES OF VISUALIZATIONS

It is our claim that although graphics has long been a tool in program development and documentation, the full power of graphics has yet to be acknowledged or exploited. We have identified ten categories of program illustrations that, together, can be of use throughout the software lifecycle. Some categories of illustrations have already been well explored with respect to programs, and the PV project was able to draw on this work directly. Other categories have been less thoroughly explored. The ten categories are:

1. System requirements diagrams
2. Program function diagrams
3. Program structure diagrams
4. Communication protocol diagrams
5. Composed and typeset program text
6. Program comments and commentaries
7. Diagrams of flow of control
8. Diagrams of structured data
9. Diagrams of persistent data
10. Diagrams of the program in the host environment

Many of these categories can apply to either <u>the program</u>, its specific <u>activations</u>, or its <u>data</u>. Moreover, illustrations can be either <u>static</u> or <u>dynamic</u>. Static illustrations portray the program at some instant of execution time, or they portray those aspects of a program which are invariant over some interval. Dynamic illustrations portray the progress of an executing program. The categories of visualization listed are discussed more fully in [8].

In earlier work [9] done in collaboration with

---

[8] Herot, C.F., Brown, G.P., Carling, R.T., Friedell, M., Kramlich, D., Baecker, R.M., An Integrated Environment for Program Visualization, in Schneider, H.J. and Wasserman, A.I. (eds.), <u>Automated Tools for Information System Design</u>, North Holland, Amsterdam (1982).

[9] Herot, C.F., Carling, R.T., Friedell, M., Kramlich, D., Design for a Program Visualization System, Technical

experts in graphic design, we described program visualization formats from a number of different categories. Under the current contract, static visualizations were created for the majority of the categories above. We chose to focus, however, on dynamic visualizations, because relatively little work had been done in that area.* Within dynamic visualizations, most of the visualizations created

---

Report CCA-81-04 (January 1981), Computer Corp. of America, Cambridge, MA.

* The pioneering work in dynamic program visualization includes:

Balzer, R.M., EXDAMS - EXtendable Debugging and Monitoring System, AFIPS Joint Spring Computer Conference (1969) 567-580.

Baecker, R.M., Two Systems which Produce Animated Representations of the Execution of Computer Programs, ACM SIGCSE Bulletin, 7, 1 (Feb. 1975) 158-167.

Dionne, M.S. and Mackworth, A.K., ANTICS: A System for Animating LISP Programs, Computer Graphics and Image Processing, 7 (1978) 105-119.

Galley, S.W. and Goldberg, R.P., Software Debugging: The Virtual Machine Approach, Proceedings: ACM Annual Conference (1974) 395-401.

Knowlton, K.C., L6: Bell Telephone Laboratories Low-Level Linked List Language, two black and white films, sound (Bell Telephone Laboratories, Murray Hill, New Jersey, 1966).

The major recent work includes:

Baecker, R.M., Sorting Out Sorting, 16mm color, sound, 25 minutes (Dynamic Graphics Project, Computer Systems Research Group, Univ. of Toronto, 1981).

Brown, M.H. and Sedgewick, R, A System for Algorithm Animation, Computer Graphics, 18, 3 (July 1984) 177-186.

Myers, B.A., Incense: A System for Displaying Data Structures, Computer Graphics, 17, 3 (July 1983), 115-126.

came from the category of depictions of structured data. Structured data presents a considerable challenge for visualization, because it takes so many forms. The PV prototype currently supports the display of simple variables, one and two dimensional arrays, linked lists, and trees. Dynamic views of flow of control were given a secondary priority because, on the whole, depictions of control flow do not need to be as varied as depictions of data. The PV prototype currently supports highlights moving through code to indicate the line being executed; much of the underlying mechanism is in place to support extension to additional views of control flow. Some photographs taken from the prototype are shown in [15].

---

[15] Kramlich, D., Brown, G.P., Carling, R.T., Herot, C.F., Program Visualization: Graphics Support for Software Development, <u>ACM/IEEE 20th Design Automation Conference</u> (June 27-29 1983) Miami Beach, Florida.

CATEGORIES OF VISUALIZATIONS

3. OVERVIEW OF THE PV ENVIRONMENT

The current workstation for the Program Visualization system consists of three medium-resolution AED 512 color displays. The user interacts with the system via a combination of data tablet with four-button puck, keyboard, and touch sensitive devices on the displays. The display arrangement is flexible, and design work was completed to move to a single high-resolution color display.

A programmer uses the PV system via a menu-oriented user interface that displays diagrams and text in multiple windows on the screen. The current PV window system is in the spirit of [17], which has been widely replicated. Information access in the PV system is aided by diagrams acting as spatial navigational aids, in the manner pioneered in [10]. The PV system has been prototyped to support programming in C, although large portions of the system are independent of the software development language. The implementation runs on a VAX 11/780 under Berkeley UNIX.

The PV environment has been designed as an "umbrella", in the sense that it is not targeted to support a single software development methodology. Basic program visualization tools can be used in the service of the programmer's chosen methodology. For this reason, the system supports the following:

- Manipulation of static and dynamic diagrams of computer systems

---

[17] Teitelman, W., A Display Oriented Programmer's Assistant, Fifth International Joint Conference on Artificial Intelligence (1977) 905-915.

[10] Herct, C.F., Carling, R.T., Friedell, M., Kramlich, D., A Prototype Spatial Data Management System, SIGGRAPH '80 Proceedings: ACM/SIGGRAPH Conference (1980) 63-70.

- Manipulation of program and documentation text

- Creation and traversal of a multi-dimensional information space

- Reuse and dissemination of tools via a library of diagram and text components (e.g., templates)

The remainder of this section contains an introduction to these facilities. The modules named are illustrated in the PV architecture diagram in Figure 1.

Manipulation of Static and Dynamic Diagrams of Computer Systems

To create, edit, and view static diagrams, the programmer/user invokes the Graphics Editor. The Graphics Editor provides a high level graphics language and gives the programmer access to prespecified diagram components (e.g., templates and collections of notational symbols) in the Library. The projected PV system would provide an optional automated visualization planning capability, performing functions such as selecting colors, sizing and positioning objects, and positioning labels. The current prototype has a limited facility for automatic sizing and positioning of objects within dynamic displays of data structures.

To create, edit, and view dynamic diagrams of programs, the programmer uses the Dynamic Object Controller. The creation of dynamic diagrams can be semiautomatic; in particular, the programmer can point to variables in his or her C code and the system will automatically select appropriate diagrams to display them. Alternatively, the user can build or select diagrams and bind diagrams to code with the Binder. In either case, when the programmer finally runs the program, the Execution Manager in the Dynamic Object Controller will monitor the running code to activate the visualization.

All diagrams are currently stored in the UNIX file system. The projected PV system would provide a design database to store diagrams and associated text.

Manipulation of Program and Documentation Text

The programmer can use the PV system to create and manipulate static text and also dynamic text (e.g., highlights moving through code to indicate flow of control).

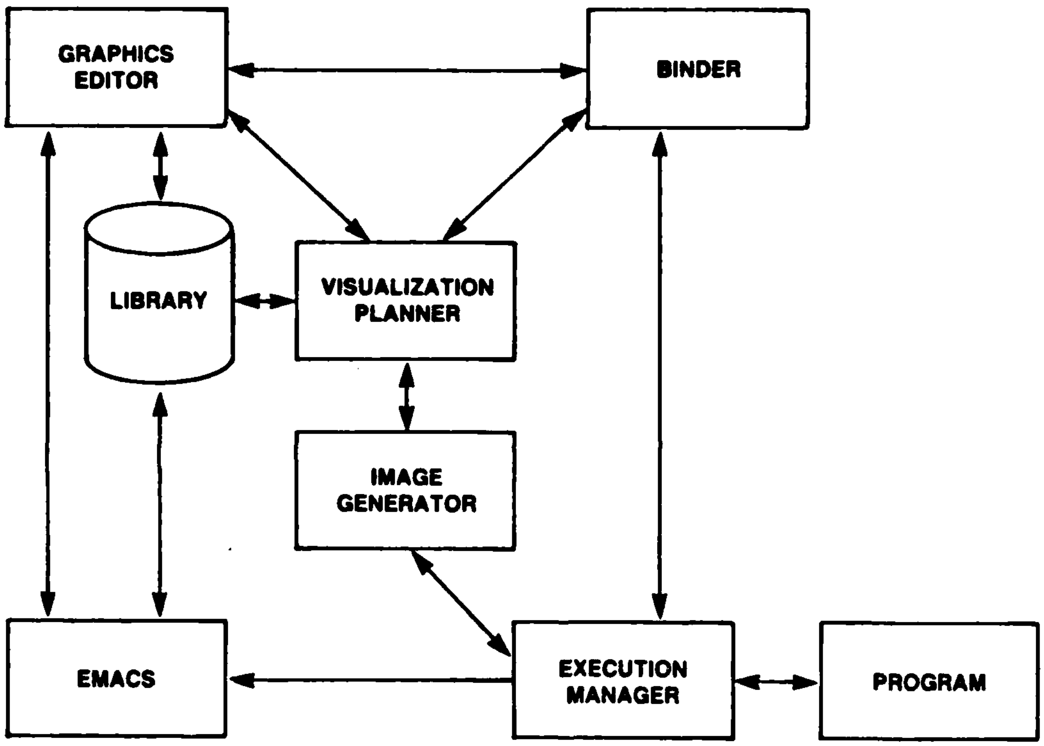


Figure 1. Architecture of the Program Visualization System.

To create and manipulate static text, the programmer invokes the Text Editor. This text editor is an implementation of EMACS enhanced for the purposes of the PV environment. EMACS was chosen because of its power and extensibility. Labeled keycaps on terminal function keys simplify the use of the text editor. Text (including code) files are stored using the UNIX file system, with the PV system providing additional spatial file access mechanisms.

To create and manipulate dynamic text, the programmer invokes the Dynamic Object Controller.

Creation and Traversal of a Multi-Dimensional Information Space

Fundamental to the programmer's ability to use the system is an information integration mechanism that allows convenient access to the quantities of diagrams and documents that are part of a large-scale system development effort. The PV system provides a multi-dimensional information space in which the programmer can establish links between graphic and textual information. For example, a user viewing a program structure diagram can "zoom-in" on program illustrations, moving from one level of abstraction to another, until at the most detailed level the program code is displayed.

Library of Diagram and Text Components (e.g., Templates)

Individual programmers or groups of programmers can add useful graphic and text components to the PV Library. The Library is a network of programming and graphics components accessible spatially, via a library organization diagram that acts as a navigational aid. Through the Library, the PV system supports efforts to standardize and share graphic notations and it supports the re-use of both code and illustration segments.

The PV system, then, is designed as a comprehensive software development environment that supports the creation and manipulation of both code and its accompanying graphic documentation. The four classes of facilities in the PV environment are discussed in more detail in each of the following four sections.

4. MANIPULATION OF STATIC AND DYNAMIC DIAGRAMS

The PV prototype supports the creation and manipulation of both static and dynamic diagrams. Each is discussed in turn.

4.1 Graphics Editor for Static Diagrams

The user begins using the graphics editor by invoking the command to create a graphics window. This results in the display of an empty rectangular background whose color, size, and location are specified by the user. The user may either work on a single window or alternate between several different windows. Windows may overlap or completely obscure each other; obscured windows are viewed via a cycle-window command that sequences through the pile of windows. The option to have diagrams larger than the size of the screen, presenting a portion at a time in a window, was designed but not implemented (due to time constraints) in the prototype system.

The graphics editor supports two methods of diagram creation:

1. drawing/specification

2. assembly

The drawing/specification method is used to create graphic objects from scratch. Assembly allows the creation of objects using pre-existing components, usually entailing considerable time savings. Drawing and specification are treated as a single method because they form a continuum. For example, specifying a straight line by pointing to the two endpoints is such a natural activity that one does not think much about the fact that the line itself is "drawn" by the graphics system rather than the user. Most of the specification currently supported in the PV prototype involves pointing to other objects to indicate relationships or pointing to locations.

The assembly method of diagram creation is simple to understand. To assemble pre-existing components, either whole components or parts of components can be copied into a diagram from other windows. In particular, components can be copied from the Library, a section of the PV information structure supporting the standardization and re-use of graphic (as well as code) components. Three types of component are used with the copy operation: building block, template, and kit. Building blocks are the simplest structures, although they may be complex visually. An example of a building block would be a graphic icon used as a label. Templates are objects with slots that the user may fill with text, or in some cases graphics. Kits are mixed collections of building blocks and templates. To take a familiar example, standard flowchart notation might be implemented as a kit. The user would then create flowcharts by copying symbols from the kit to the diagram window, filling text in slots as necessary. Connectors would be created by selecting a connector symbol from the kit and specifying the end points of the connector.* If the user wished to revise a flowchart diagram, he or she could use diagram editing commands described later in this section.

For drawing/specification, the system's main model of graphic objects is structural, i.e., it represents graphic components as named entities that have parts and subparts. The system also supports a secondary, vector model used for detail drawing to give specific visual or symbolic characteristics of an object. Finally, a third graphic model, the character model, is supported for text in the form of labels and short comments. The difference between the structural and the vector model becomes clear if one thinks of four lines drawn to form a square. In the vector model, if a person points to a place inside the square and close to one of the lines, we would assume a reference is being made to that closest line. In the structural model, pointing to this same place (or to anywhere within the square) would constitute a reference to the object as

* In the prototype, all connector drawing is done through commands on the menu. The ability to include connectors as separate symbols in kits was not implemented as part of the prototype, although it is clearly desirable when different line colors, weights, and textures are available as options. Further work in this direction, along with work on automatic routing of complex connectors, would be desirable.

a whole. In the vector model, a move command would result in one line moved away from the other three. In the structural model, all four lines and the space they enclose would move.

For the structural model, the current PV prototype supports the creation of rectangles, circles, and polygons.* The user designates an object as part of another by pointing to the parent object at the time the object is created. This explicit reference allows parts to overlap or to be partially outside the boundaries of their parent part. This is a useful feature when objects are first created, to allow the user to experiment freely with part sizes. The user designates connectors between objects by either drawing them directly or selecting an option that causes the system to draw straight line connectors for the user. Note that connectors are logical constructs rather than purely graphic ones. When an object is moved, any of its associated system-generated connectors are redrawn to reflect the new position. A final type of object in the structural model is the "slot", used for constructing templates. Slots are rectangular objects that the user creates by specifying two points and giving a name.

For the vector model, the PV prototype supports the creation of lines of different weights, circles, and filled regions. For the character model, the system supports a choice of text fonts (MIT shaded fonts, Berkeley fonts, and those available on the host hardware). For all models, graphic object construction is aided by user-specified rectangular grids, both global grids associated with entire diagrams and local grids associated with individual parts. Grids are specified by either pointing to two points (to indicate the box size) or keying in the number of vertical and horizontal divisions of the space. Color mixing is supported in the RGB model by selecting positions on three color bars.

---

* Support for polygons is somewhat limited in the PV prototype because the hardware that we were using did not support polygon generation in the firmware. Since firmware support for polygons is now common in graphics hardware, duplication of this work in software was given low priority.

Besides creation of graphic objects, the system supports manipulation operations: copy, move, resize, delete, and recolor.* These operations have two options. Choosing one option causes the operation to apply only to the part the user has pointed to; choosing the other option causes the operation to apply to the part and all its subparts recursively. Although both options apply to each operation, the default differs. For example, it is more common to recolor a single part than to recursively recolor (in a single color). Alternatively, it is more common to delete recursively than to delete a containing part out from under its subparts. (For simple parts, of course, deletion behaves the same either way.)

A set of dual-option commands paralleling the set of object manipulation commands is provided for graphic text. Thus, blocks of graphic text may be manipulated either individually or recursively. The current unit of text that can be manipulated in this way is defined with respect to an object or part, although extension to a lower level of granularity may be desirable.

In addition to creating and manipulating graphic objects, the user can create links between graphics and text or between two levels of graphics. These links are discussed in Section 6. An object-info command displays the status of each object: name, graphic properties, and existing links. When the user wishes to stop editing, diagrams can be named and saved for later use.

---

* The structural and vector models are represented by two parallel sets of manipulation commands. The PV prototype fully implemented the manipulation commands for the structural model. There are some gaps in coverage for manipulation commands in the vector model, although the basic create and move commands are in place. Support for the vector model within PV was given low priority because we were able to use a separate graphic editor previously implemented at CCA. Note that a single set of commands could be used for both models if two different modes were introduced. The use of parallel command sets was helpful for development, since we did not know if the sets would ultimately need to diverge. The need to treat structural and vector manipulation separately is obvious from the example of the square cited above. In that case, pointing to the same place could mean two different referents according to which model the user is assuming.

4.2 Creation and Control of Dynamic Diagrams

The project focused on dynamic views of structured data, due to the challenge presented by the many forms that data can take. The types of dynamic graphics supported in the prototype are:

1. in-place updates of values
2. indicators moving on vertical or horizontal scales
3. data cells created, rearranged, and deleted to reflect different pointer relationships

To specify dynamic graphics, the user must supply enough information for the system to establish correspondences between the code and the static graphic components of the dynamic visualization. We refer to this process as "binding". There are two binding methods, paralleling the two methods for static object creation: the user may either draw/specify the binding from scratch or he or she may assemble the binding by combining pre-bound components.

To create a binding from scratch, the user first creates any necessary static components using techniques described above. Taking a simple example, a user wishing to display an integer might create a special cell that shows the variable name and indicates the type and precision. Once the necessary static components have been created, the user accesses a pop-up menu of dynamic types and selects the type desired. In the current prototype, binding for each dynamic type is done by a discrete subroutine which asks the user questions appropriate to the type. Questions relate to the type, range, and related properties of a data structure as well as the graphic regions in which information is to appear. While the discrete subroutine approach was useful for the early stages of the work, it is desirable to replace the fixed binding protocols by more flexible interactions. A forms-oriented interface would allow the user to order the interactions, and it would permit more sharing of binding code across dynamic types.

Once the user has finished the binding for each component in the visualization, this information remains associated with the graphic objects. This not only permits dynamic visualization specifications to be saved for re-use, but it also permits dynamic graphic components to be "packaged" so that they can be used in other assemblies. Thus, creation by assembly works for dynamic as well as static graphics. This supports the cluster capability described in Section 7, whereby a code template and a corresponding graphic template can be pre-defined (including dynamics) and then inserted in the code and the visualization, respectively. To complete that part of the visualization, the user need only fill in specifics, such as the name of the variable. Use of such a cluster to do binding "by assembly" is illustrated in the videotape that accompanies this report.

To view dynamic visualizations once they have been created, the system provides both speed control and stepping. Viewing speed is controlled by a bar on the menu, which the user can lengthen or shorten to get different percentages of the maximum speed. Since absolute times are not particularly meaningful in this context, we chose to mark only quarters on the bar, although finer gradations may be selected. For stepping, the visualization is displayed one step at a time, in the lowest level of granularit currently displayed. Thus, if program code is displayed, the system behaves like the standard step mode and executes one line, waiting for a user signal (in this case a button-push) to continue. If, however, only a visualization of one data structure is displayed, then the pauses come only each time the data structure is updated. This is an important capability, because it lets the user move from very general view of the program to very localized view merely by selecting different diagrams. It also permits the user to view program execution from the perspective of selected data structures, either key data structures or those that are of current, temporary interest for debugging.

The PV prototype, then, embodies a representative set of dynamic types that can be used for common data structures such as numeric variables, arrays, linked lists, and trees. This area deserves further research, both to extend the coverage of dynamic types and to increase the sophistication of the animation techniques available. Besides visual polish, more sophisticated animation techniques can add considerably to the clarity and comprehensibility of the dynamic visualization, particularly for large data structures.

5. DISPLAY AND MANIPULATION OF TEXT

Due to the magnitude of the task of PV system design and implementation, we made a pragmatic decision to incorporate existing non-graphic software development tools. In the case of text editing, this decision lead to a workable arrangment, although one with less integration of graphics and text than we would have liked. This section briefly describes text handling within the PV prototype, followed by some comments on desirable directions for the future.

The text editor chosen for the PV prototype was EMACS, in the form of CCA's EMACS implementation for Berkeley UNIX. EMACS was selected because of its power and because of the support it gives programmers in customizing their editing environment. The PV project funded extensions to CCA EMACS to achieve a closer coupling between text and graphics. One such extension allowed extraction of enough information from C code to permit a simple level of pattern matching on variable declarations. This pattern matching was used in automatic access of a graphic depiction appropriate to the variable type. Another extension was the addition of EMACS "picture mode", which permits the user to manipulate text as a two-dimensional object. This mode provides an alternative to the line-oriented "one-dimensional" mode that is standard to EMACS. From the PV user's point of view, the two dimensional model is closer to the model used for graphics, and it is useful for certain types of editing.

In the PV prototype, text is displayed in three types of windows:

1. EMACS

2. UNIX shell

3. read-only text or code

EMACS windows allow the text to be edited, with viewing controlled by the standard EMACS commands to page through files. Text is currently stored in the UNIX file system. Shell windows display the UNIX C-shell command prompt and permit users to execute the full range of commands and application programs that have non-graphic output. Finally, read-only windows are used for temporary displays of text or code displayed for the user's information or for the user to verify.

One use of EMACS windows is in the display of dynamic text, e.g., code with moving graphic indicators (highlights or arrows) to indicate the progress of the execution of a program. The current prototype supports the dynamic display of one level of code in a window, although extension to show subroutine execution is possible. Once dynamic text is extended to multiple windows, it would be straightforward to couple the text display with a stack diagram and a highlighted program structure diagram. The PV prototype provides the basic framework to support these extensions (see Section 8).

We mentioned that integration of graphics and text, while acceptable, is less extensive than we would have liked. One capability that would be desirable is the ability to assign a fuller range of graphic attributes to program text, which has been explored by Baecker and Marcus [2]. Another desirable capability is structure-editing for both code and formatted text. Incorporation of a structure editor in the spirit of [16] would be particularly useful for handling templates and for specifying pattern matches for automatic library accesses (e.g., find a graphic depiction that matches information in a variable declaration).

---

[2] Baecker, R.M. and Marcus, A., On Enhancing the Interface to the Source Code of Computer Programs, Human Factors in Computing Systems, CHI83 Conference Proceedings, Association for Computing Machinery, New York (1983) 251-255.

[16] Teitelbaum, R.T., The Cornell Program Synthesizer: A Microcomputer Implementation of PL/CS, TR 79-370 (1979), Department of Computer Science, Cornell Univ.

6. MULTI-DIMENSIONAL INFORMATION SPACE

The multi-dimensional information space provided by the PV system permits the programmer to establish links: diagram to diagram, text to text, diagram to text, and text to diagram. Links are associated with the whole or with individual parts of the diagrams or text. These links are then traversed by the user by selecting a "zoom" command and then pointing to the relevant object, object part, text file or text section.

Links are created by the user by selecting from a menu of relationships. The user then points to the objects that participate in that relationship or, if a destination object is not currently shown, its name is keyed in. The relationships currently supported within the PV prototype are:

1. source: graphics to code
2. object: graphics, text or code to graphics
3. documentation: graphics or code to text
4. notes: anything to text

uther code to code relationships were implemented for experimentation. In the prototype, the user follows links by selecting a command of the appropriate link type and then pointing to an item (graphic, text or code) displayed on the screen. One special command, "zoom", is intended for following object links that relate items at two different levels of detail.

In keeping with PV's role as an umbrella system, a desirable (and straightforward) extension would be to allow the user to define new binary link types. Another desirable extension is support for simple type checking of link parameters when the links are established.

The links provided by the PV system are intensionally general enough that they make few constraints on the type of information relationships that can be specified. In working with the PV prototype to build examples, however, we followed the approach used in SDMS [10]. In our examples, the dominant information organization is hierarchic. Other non-hierarchic relationships are then used as needed. That is, the dominant motion through the information space is "zooming in" on graphic objects to get a more detailed view of a particular section.

For the PV prototype, we suggested a core set of four hierarchies: system requirements, structure, evolution (i.e., version control), and the library. The first three of these hierarchies are project-specific collections of information about the system being developed by the user. The fourth hierarchy, which is shared by all users of the PV system, is described in the next section. Note that the project-specific hierarchies are suggested as defaults, and additions corresponding to any of the steps in the software lifecycle would be appropriate.

We have called the top level diagram for each hierarchy the navigational aid, or navaid. This term is borrowed from SDMS, where navaids present visual context for detailed views and they give a means for moving around an information space to select more detailed views. Although the PV navaid is not exactly equivalent to the SDMS one, we use the term to emphasize the role of the top level diagrams as a map of their respective information spaces. For example, the system structure navaid might show a top level structure diagram in the user's prefered graphical notation. A more detailed diagram is then associated with each module, with this process continuing to the depth needed. At the most detailed level the user might then attach the actual code. The navaid diagram can serve as a starting point for access to any part of the structure hierarchy, and so it acts as a global map.

To keep the user from getting lost as he or she moves between levels of the hierarchies, there is also a need for immediate context. Clear diagram labeling and numbering schemes are important aids; the project designed graphic icons for the suggested hierarchies so that each window could be labeled with the hierarchy to which it belonged. Again applying techniques from SDMS, we found it helpful to display with each diagram a miniaturized view of the diagram above it in the hierarchy. The

---

[10] Herot, C.F., Carling, R.T., Friedell, M., Kramlich, D., A Prototype Spatial Data Management System, SIGGRAPH

miniaturization has a "you are here" region marked, and a set of bars on the side show the current depth in the hierarchy.* One question that needs to be investigated is whether these miniaturized views should only reflect static hierarchic relationships or whether they should instead reflect the path that the user took to access the information during the current session. That is, if access was via a cross-hierarchy link rather than via zooming, is it most useful to see a miniaturization of the hierarchic context or the access context.

In summary, the PV system provides a multi-dimensional information space that aims to be neutral with respect to the information structures defined. The suggested mode of use, however, is to construct a set of hierarchies, interrelated by cross-links as necessary.

---

* A small amount of additional code would be necessary to completely support this feature in the prototype.

7. LIBRARY OF DIAGRAM AND TEXT COMPONENTS

The PV Library is a collection of code, text, symbols, and diagrams shared by users of the PV system. It is the repository for code components (e.g., code templates) for individual projects as well as for general design notations shared by projects. Graphical components stored in the Library allow PV users to construct many types of program visualizations simply by combining elements rather than by starting from scratch.

7.1 Types of Components in the Library

Four basic types of components are stored in the PV Library:

1. building blocks
2. templates
3. kits
4. generators

Building blocks are fully instantiated objects or complete modules. A code example of a building block would be a subroutine to compute square roots. A graphic example would be a symbol used as an iconic label, for example the icons used to label classes of commands on the PV menus.

Templates are objects with internal slots that must be filled in by the user (or the system) to complete the specification of the object. The initial PV prototype does not support checking of information placed in slots, but it does support automatic propagation of slot entries so that the user is not required to fill in information more than once. This is discussed in more detail below.

Kits are sets of templates and building blocks, e.g., standard flowchart notation could be represented within a kit.

Generators can be though of as general tools and application programs that produce objects, either directly or interactively. Examples of generators are text formatters, code formatters, and graphic menu building programs. The difference between generators and subroutines that are used as building blocks is that building blocks are used directly by the PV user, while generators are run to create objects that are used.

The PV prototype as implemented supports the first three types of components. Generators can be stored in the Library and they can be run within UNIX shell windows; they cannot, however, be automatically invoked when the Library node is accessed.

To support the user in carrying out related parallel tasks, the PV Library permits components to be grouped into "clusters". For example, the user may want to construct graphic visualizations, write code, and create the appropriate textual documentation all at the same time. To permit the application of assembly techniques to these tasks, clusters of components may be stored in the Library. For example, code for a numeric subroutine, its dynamic visualization, and its manual page may be stored as three building block components in a cluster in the library. The members of the cluster may either be accessed individually, or they may be accessed by a special automatic method discussed at the end of this section. Finally, note that when templates participate in clusters, automatic propagation of slot-fillers is supported across templates. For example, the variable name slot in a data declaration template might be linked to a name slot on a graphic depiction of the data type. Filling in the text in one or the other slot can lead to automatic propagation of the text to the related slot.

7.2 Overview of Library Organization

While the PV prototype does contain some specialized code related to the Library, the Library is basically implemented as an application of the information linking techniques described in Section 6. As such, Library organization is relatively open-ended. We do, however, suggest some methods of organization that we expect to be of use when large numbers of components are added to the Library.

The proposed Library organization consists of three taxonomies:

1. general programming concepts

2. graphical structures

3. projects

A component in the Library may be a descendant of any one, or any combination of, the three major taxonomies. The organization of each taxonomy is discussed in turn.

An attractive candidate for the taxonomy of general programming concepts is the AFIPS Taxonomy of Computer Science and Engineering [14]. This taxonomy contains about 1500-2000 nodes under the major headings:

1. hardware

2. computer systems

3. data

4. software

5. mathematics of computing

6. theory of computation

7. methodologies

8. applications/techniques (illustrative)

9. computing milieux

The taxonomy is a tree with cross-references that is at

---

[14] Taxonomy of Computer Science and Engineering, AFIPS Taxonomy Committee, AFIPS Press, Arlington, VA (1980).

most six nodes deep. Some additions to this taxonomy would be necessary to reflect technological change since its publication, and some categories (e.g., computing milieux) might be pruned as irrelevant to the PV environment. On the whole, however, it appears that this taxonomy can provide a useful framework for organizing PV components according to their relevance to programming.

To organize components according to their graphic structure, we propose to use an approach described by Twyman [15]. Twyman lists what he calls "methods of configuration" by which he means "the graphic organization or structure of a message which influences and perhaps determines the 'searching', 'reading', and 'looking' strategies adopted by the user" (p.120). Twyman proposes the following categories, for which we have added examples relevant to programming:

1. Pure Linear
   pure examples are rare, but some depictions of strings belong to this class

2. Linear Interrupted
   e.g., connected text

3. List
   e.g., certain types of graphic pop-up menus

4. Linear Branching
   e.g., depictions of tree data structures with nodes and connectors drawn; also, indented code

5. Matrix
   e.g., tables with discrete alphanumeric elements such as two-dimensional arrays

6. Non-Linear Directed Viewing
   e.g., network diagrams

7. Non-Linear, Most Options Open
   e.g., two-dimensional memory maps

A second dimension suggested by Twyman, the "Mode of

---

[15] Twyman, M., A Schema for the Study of Graphic Language, in Processing of Visible Language I., 117-150.

Symbolization" continuum (verbal/numerical, pictorial and verbal/numerical, pictorial, and schematic), is relevant for PV as well. It can be used as a secondary classification scheme within method of configuration. It might also be useful to subclassify by whether a component is dynamic, and, if so, what type of dynamics is used.

Finally, project-specific entries may be made and indexed under the third proposed PV taxonomy. The project taxonomy allows a group to identify a set of components, both standard technical tools (e.g., Jackson diagrams, etc.) and special-purpose components constructed for the group. The structure of the project taxonomy is up to individual projects. Projects can add intermediate classifying nodes to the Library information structure, so that the organization of components for a given project is as simple or complex as desired.

7.3 Library Access

The Library is available to the user as soon as the PV environment is invoked. There are three ways to access components in the Library: by navaid, by keyboard, and automatically from a cluster entry. Each is discussed in turn.

The main method of PV Library access is spatial, via the Library navaid. The Library navaid, like the other navaids proposed for the PV system, is a standard PV diagram in a standard PV window. The navaid displays the organization of the Library as a lattice, with nodes for each component, cluster, and classifying category. Graphic icons may supplement text to label major classifying nodes. A more limited example navaid constructed for the prototype is a tree with nodes connected by straight line connectors. The user moves around the navaid by the standard PV display command set. This set includes a command to move an entire window-full in any of four directions relative to the current window, as well as a "goto" command that takes a node name as argument and centers the window on that node. Note that motion over navaids requires support for diagrams larger than a window; although this capability was designed, it was not implemented in the prototype system.

Once the user has chosen a component node, he or she can zoom in on it to get a view of the component itself. The zoom command has two options: to display the component in the current window, replacing the navaid, or to create a separate window for viewing the component. In either case, components can then be copied from the detail window to an editing window. This copying operation actually creates an instance copy, which the user can augment or modify as desired.

A second way to access Library components is directly, by keyboard. Since the Library is implemented as a collection of standard objects linked via the standard information structure, the user can execute a "use" command and key in the unique identifier of the component. While these identifiers would not necessarily be easy to remember (hence our emphasis on spatial access via navaid), they are accessible to the user by executing an "object-info" command on the component detail window. For frequently accessed components, this direct approach can be efficient.

The third way that library components are accessed is automatically by the PV system itself. This is done to pick up components within clusters, e.g., visualizations associated with data structure types. First, an element in a cluster is accessed by either of the two means described above, and the component is copied to the target window. If the component is a template, text slots in the template may be filled in by the user. The user then selects a special "autoaccess" button from the PV menu. The system goes to the Library, retrieving other elements in the cluster. In the current implementation, autoaccess is restricted to clusters of two elements (code and visualization), but extension to larger clusters would be straightforward. The system would need to display the types of the information links used to construct the cluster, and the user would then choose the type, and hence the component, of interest. When autoaccess finds the component desired, it returns it with any necessary slot filler propagations done. The component is shown in a temporary window, and, if the user is satisfied with the selection, a button-push causes the component to be inserted into the editting window. This automatic access sequence is illustrated in the PV videotape that accompanies this report.

In summary, the PV Library currently accomodates three kinds of access: spatial, direct, and automatic access within clusters.

8. MONITORING RUNNING PROGRAMS

In order for program visualization to be a practical technique for monitoring large programs, the monitoring must be non-intrusive. That is, visualization of compiled code must be done without recourse to the inclusion of special graphics statements into the source code. We describe here techniques that can be applied to monitor programs (other than real-time systems) that run under the UNIX operating system. Our discussion centers on the Execution Manager module shown in Figure 1.

The Execution Manager, as its name implies, interacts with and manages the program being monitored. It has two main tasks: determine where the program is currently executing, and monitor user-selected variables for updates.

Our implementation of the Execution Manager makes use of software debugging facilities provided in the UNIX operating system and hardware features used to implement virtual memory. The UNIX environment provides a set of facilities by which one process may manipulate the registers and address space of another process. Several new features were added to allow the monitoring process (Execution Manager) to manipulate the memory mapping of the monitored process (the user's program). In addition, the Portable C Compiler was modified to produce a supplemented symbol table, which the Execution Manager uses to locate variables and code in the user's program.

There were two primary goals in the implementation of the Execution Manager: minimize any special requirements on the monitored program, and make monitoring as efficient as possible.

8.1 Minimizing Special Requirements

The first goal was achieved by placing the burden of program monitoring on the C compiler and the Execution Manager. The modified C compiler produces a complete description of the global variables, procedures, local variables, and source code to machine code mapping. The Execution Manager uses this augmented symbol table to map symbol names into addresses in the monitored process. It should be noted that the C compiler is a modified version of the one distributed by U.C. Berkeley. Berkeley's version produces a very complete symbol table; the compiler was modified to correct the few deficiencies that it had.

In order to monitor a particular program, the only requirements are that it be compiled by the modified C compiler and that it be linked with a special assist routine described below. The code generated by the modified C compiler is identical to that generated by the conventional compiler. That is, there are no performance penalties in using the modified compiler. The only difference is the larger symbol table that is produced.

8.2 Efficiency

The second goal, efficiency in program monitoring, was supported by exploiting applicable hardware characteristics. Two of the characteristics exploited are found on nearly all machines: the single-step execution mode and the breakpoint instruction. The third characteristic is virtual memory, a feature common on new machines.

The single-step mode of execution is used principally to track the execution of the monitored process. After each machine instruction has executed, the process traps to the operating system which then notifies the Execution Manager. The Execution Manager reads the current value of the program counter and maps the value into a file name, procedure name, and line number in the source code using the mapping provided by the compiler in the symbol table. This information then can be used for highlighting a line of code in a source code display or highlighting a procedure invocation on a stack.

The breakpoint instruction also is used to track execution, but at a coarser granularity. Although the user may insert breakpoints anywhere in the code, the most common use of the breakpoint is at the beginning of a procedure that contains local variables to be monitored.

Our use of virtual memory is less self-evident, and we spend the rest of this section discussing it. In order to maintain an accurate display of data structures which the user has selected for viewing, the Execution Manager must detect any updates. There are several ways in which this detection might be done: analyze the executing program for all assignments to the selected data structures; examine each data structure after every instruction for value changes; tag the data structures so that an update will cause an event. These approaches are discussed below.

Code analysis will catch most, but not all, assignments. Programs written in languages that allow indirect references through pointers (as in C) or which allow the dynamic creation of data structures at run-time (through memory allocation routines) may have "hidden" assignments in them. Examining each data structure after every instruction is very inefficient and slows the speed of execution of the monitored process significantly. Tagging the data structures is the most general approach, but it suffers from several limitations: most machines do not allow individual memory locations to be tagged; local variables pose a problem because their locations in memory are not fixed; and register variables usually cannot be tagged.

To overcome these limitations, the PV implementation uses a variation on tagged memory. Instead of tagging individual memory locations, an entire page is tagged. By setting the protection of a page that contains a data structure being monitored to be read-only, a trap will occur whenever a write occurs on that page. The Execution Manager then checks to see whether the address(es) just written include a data structure which it is monitoring. If so, the new value is read and displayed.

The process of catching updates works in a straightforward way for global and static variables because their addresses are fixed at load time and do not change. However, variables local to a procedure are allocated space on the stack. Thus their real addresses will vary, depending on the history of procedure invocations. To overcome this problem, the Execution Manager sets a special breakpoint at the beginning of each procedure that contains local variables to be monitored. When one of these breakpoints is executed, the following steps are performed:

1. Record the value of the stack pointer at that point.

2. Copy the current stack frame (corresponding to the procedure just invoked) on the stack to make room for a new interposed stack frame. This new frame represents the context of the special assist routine mentioned earlier and is used during the cleanup after the watched procedure returns.

3. Set the stack page(s) that contain the local variables to be read-only.

4. Resume execution of the interrupted procedure.

Execution continues as in the case of global variables. Traps will occur when the protected stack pages are written to. When the procedure returns, it returns to the special assist routine. The assist routine signals the Execution Manager that a procedure that had local variables being monitored has returned. The Execution Manager then resets the protection on the pages and resumes execution of the monitored program.

The implementation of the Execution Manager has been described. The Execution Manager monitors the status of the executing program in such a way as to minimize the effects of the monitoring on performance. This is achieved by means of modifications to the C compiler used to compile the monitored programs and by extending the UNIX kernel to allow one process access to another process' memory map.

9. CONCLUSIONS

We expect on-line, interactive graphics to have a profound impact on software development, analogous to the way that word processors have affected the production of text. With respect to static graphics, the multidimensional graphic information structure that we have developed for PV will allow the programmer to move easily between pieces of related information (e.g. requirements and system structure). With respect to dynamic graphics, PV's animated views of program execution can help programmers achieve a deeper and more accurate understanding of the behavior of their programs.

The work done under this contract has led to a better understanding of the role of diagrams, particularly dynamic diagrams, in the software development lifecycle. Because the research area is relatively new, however, considerable work remains to be done. One challenge is the graphical depiction of very large data structures. The PV graphic representation supports multiple levels of detail so that users can view more detail as needed. The requisite support for data abstraction is not present within C, however, so that this technique can be best explored in the context of a language such as Ada.

Another challenging problem is the controlled use of dynamic graphics. The PV project has explored techniques for drawing the user's attention to parts of the display that are about to change. More work must be be done, however, to insure that dynamics enhance the clarity of the visualization. A more complete dynamic vocabulary appropriate to programming needs to be developed, with particular attention paid to the demands of visualizations (both graphic and text) that are so large that they can only be viewed in segments.

CONCLUSIONS

10. REFERENCES

[1] Balzer, R.M., EXDAMS - EXtendable Debugging and Monitoring System, AFIPS Joint Spring Computer Conference (1969) 567-580.

[2] Baecker, R.M. and Marcus, A., On Enhancing the Interface to the Source Code of Computer Programs, Human Factors in Computing Systems, CHI83 Conference Proceedings, Association for Computing Machinery, New York (1983) 251-255.

[3] Baecker, R.M., Sorting Out Sorting, 16mm color, sound, 25 minutes (Dynamic Graphics Project, Computer Systems Research Group, Univ. of Toronto, 1981).

[4] Baecker, R.M., Two Systems which Produce Animated Representations of the Execution of Computer Programs, ACM SIGCSE Bulletin, 7, 1 (Feb. 1975) 158-167.

[5] Brown, M.H. and Sedgewick, R, A System for Algorithm Animation, Computer Graphics, 18, 3 (July 1984) 177-186.

[6] Dionne, M.S. and Mackworth, A.K., ANTICS: A System for Animating LISP Programs, Computer Graphics and Image Processing, 7 (1978) 105-119.

[7] Galley, S.W. and Goldberg, R.P., Software Debugging: The Virtual Machine Approach, Proceedings: ACM Annual Conference (1974) 395-401.

[8] Herot, C.F., Brown, G.P., Carling, R.T., Friedell, M., Kramlich, D., Baecker, R.M., An Integrated Environment for Program Visualization, in Schneider, H.J. and Wasserman, A.I. (eds.), Automated Tools for Information System Design, North Holland, Amsterdam (1982).

[9] Herot, C.F., Carling, R.T., Friedell, M., Kramlich, D., Design for a Program Visualization System, Technical Report CCA-81-04 (January 1981), Computer Corp. of America, Cambridge, MA.

[10] Herot, C.F., Carling, R.T., Friedell, M., Kramlich, D., A Prototype Spatial Data Management System, SIGGRAPH '80 Proceedings: ACM/SIGGRAPH Conference (1980) 63-70.

[11] Kramlich, D., Brown, G.P., Carling, R.T., Herot, C.F., Program Visualization: Graphics Support for Scftware Development, ACM/IEEE 20th Design Automation Conference (June 27-29 1983) Miami Beach, Florida.

[12] Knowlton, K.C., L6: Bell Telephone Laboratories Low-Level Linked List Language, two black and white films, sound (Bell Telephone Laboratories, Murray Hill, New Jersey, 1966).

[13] Myers, B.A., Incense: A System for Displaying Data Structures, Computer Graphics, 17, 3 (July 1983), 115-126.

[14] Taxonomy of Computer Science and Engineering, AFIPS Taxonomy Committee, AFIPS Press, Arlington, VA (1980).

[15] Twyman, M., A Schema for the Study of Graphic Language, in Processing of Visible Language I., 117-150.

[16] Teitelbaum, R.T., The Cornell .. gram Synthesizer: A Microcomputer Implementation of PL/CS, TR 79-370 (1979), Department of Computer Science, Cornell Univ.

[17] Teitelman, W., A Display Oriented Programmer's Assistant, Fifth International Joint Conference on Artificial Intelligence (1977) 905-915.

11. PUBLICATIONS AND MAJOR PRESENTATIONS

December 1981

Christopher Herot, Mark Friedell, and Diane Smith took part in a DARPA conference organized by Craig Fields of the System Sciences Division. Copies of the presentations were compiled in "DARPA Conference on Computer Software Graphics", Key West Florida, Dec. 13-15 1981.

January 1982

Christopher Herot and Gretchen Brown participated in the IFIP WG 8.1 Working Conference on Automated Tools for Information Systems Design and Development, New Orleans, 26-28 January, 1982. The paper presented at this conference, "An Integrated Environment for Program Visualization", appeared in Schneider and Wasserman (eds.), Automated Tools for Information Systems Design, North-Holland Publishing Co., 1982.

February 1982

Christopher Herot gave a presentation on PV at a meeting of the Northeastern ACM Chapter on February 18.

March 1983

Gretchen Brown was a panelist at the ACM SIGSOFT/SIGPLAN Symposium on High-Level Debugging. The position paper for this workship appeared as: Christopher F. Herot, David Kramlich, Richard T. Carling, Gretchen P. Brown Debugging in an Integrated Graphics Programming Environment, Preprint of the Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on High-Level Debugging, 1983 March 20-23, Pacific Grove, California.

May 1983

Christopher Herot, David Kramlich, and Paul Souza (graphic designer affiliated with WGBH) participated in the DARPA Program Visualization Conference, organized by Clinton Kelly of the System Sciences Division.

PUBLICATIONS AND MAJOR PRESENTATIONS

June 1983

David Kramlich presented a paper at the ACM/IEEE Design Automation conference, which appeared as: David Kramlich, Gretchen P. Brown, Richard T. Carling, Christopher F. Herot Program Visualization: Graphics Support for Software Development, ACM/IEEE 20th Design Automation Conference, June 27-29 1983, Miami Beach, Florida.

DISTRIBUTION LIST

Defense Documentation Center Cameron Station Alexandria, VA 22314	12 copies
Office of Naval Research Arlington, VA 22217	
Information Systems Program (437)	2 copies
Code 200	1 copy
Code 455	1 copy
Code 458	1 copy
Office of Naval Research Eastern/Central Regional Office Bldg. 114 Section D 666 Summer Street Boston, MA 02210	1 copy
Office of Naval Research Branch Office, Chicago 536 South Clark Street Chicago, IL 60605	1 copy
Office of Naval Research Western Regional Office 1030 East Green Street Pasadena, CA 91106	1 copy
Naval Research Laboratory Technical Information Division, Code 2627 Washington, DC 20375	6 copies
Dr. A. L. Slafkosky Scientific Advisor Commandant of the Marine Corps (RD-1) Washington, DC 20380	1 copy
Naval Ocean Systems Center Advanced Software Technology Division Code 5200 San Diego, CA 92152	1 copy
Mr. E. H. Gleissner Naval Ship Research & Development Center Computation & Mathematics Department Bethesda, MD 20084	1 copy

Capt. Grace M. Hopper (008)
Naval Data Automation Command — 1 copy
Washington Navy Yard
Bldg. 166
Washington, DC 20374

Defense Advanced Research Projects Agency
Attn: Program Management/MIS — 3 copies
1400 Wilson Boulevard
Arlington, VA 22209

Mr. Robert J. Power — 1 copy
DCRB-DCB-B8
Administrative Contracting Officer
Defense Contract Administration Services
Management Area, Boston
495 Summer Street
Boston, MA 02210

END

FILMED

1 84

DTIC